2
Coleta de Lixo

Neste capítulo descrevemos de forma sucinta os algoritmos clássicos usados na implementação de coletores de lixo. O objetivo básico desta breve revisão é facilitar a compreensão de alguns tópicos que serão abordados nos capítulos seguintes. O leitor interessado em uma descrição mais detalhada de técnicas de gerenciamento automático de memória deve consultar os excelentes trabalhos de Wilson [69], e Jones e Lins [41].

A função básica de um coletor de lixo é reciclar blocos de memória heap que contenham dados que não vão mais ser usados pela aplicação, proporcionando portanto uma maior capacidade de alocação e utilização de memória. Do ponto de vista operacional, blocos recicláveis correspondem a objetos alocados dinamicamente que não podem, ou simplesmente não vão mais ser acessados pelo programa cliente. Para descobrir quais objetos podem ser coletados é preciso conhecer quais endereços de memória contém ponteiros, e quais destes ponteiros serão dereferenciados pelo programa cliente. Infelizmente o problema de determinar com precisão quais ponteiros serão efetivamente dereferenciados por um programa qualquer é complexo, e nem sempre tem solução [50]. Por conseguinte, coletores de lixo reais usam técnicas que permitem encontrar *soluções aproximadas* para este problema, sem incorrer em custos de processamento excessivos.

Para garantir a correção da aplicação, objetos que ainda serão acessados pelo programa cliente não podem ser coletados sob nenhuma hipótese. Consequentemente, qualquer técnica de aproximação usada por coletores de lixo precisa ser conservadora, ou seja, nenhum objeto que vai ser acessado pelo programa cliente pode ser coletado, mas alguns objetos que não vão ser mais acessados podem eventualmente deixar de ser coletados.

Na prática, duas técnicas básicas são usadas para determinar o conjunto de objetos coletáveis em uma aplicação qualquer:

- Na *contagem de referências*, para cada objeto alocado na memória heap é associado um contador que indica o número de ponteiros que apontam

para o objeto. Se a contagem de referências atinge zero, não existem mais ponteiros para o objeto, e o mesmo pode então ser coletado.

- Na técnica de *rastreamento* (tracing), o conjunto de objetos que podem ser acessados pelo programa cliente é determinado percorrendo-se o caminho formado pelas referências e ponteiros entre objetos (o *grafo de conectividade*). Como ponto de partida desta varredura são usados todos os pontos de acesso do programa cliente para a memória heap. Esse conjunto de pontos é denominado *conjunto-raiz*, e pode incluir variáveis globais, variáveis e valores alocados na pilha de execução, e endereços armazenados em registradores. A princípio, todos os objetos alocados na memória heap que não pertencem ao grafo de conectividade não podem mais ser acessados pelo programa cliente, e portanto são considerados coletáveis.

A Figura 2.1 apresenta um exemplo de um grafo de conectividade, onde o conjunto-raiz é representado pelos objetos A, B, C e D, e referências são indicadas por setas. Percorrendo-se a cadeia de referências que tem origem no conjunto-raiz é possível alcançar, dentre outros, o objeto E, que portanto não é coletável. Já os objetos F, G, H, I e J não são acessíveis a partir do conjunto-raiz, e podem ser coletados imediatamente.

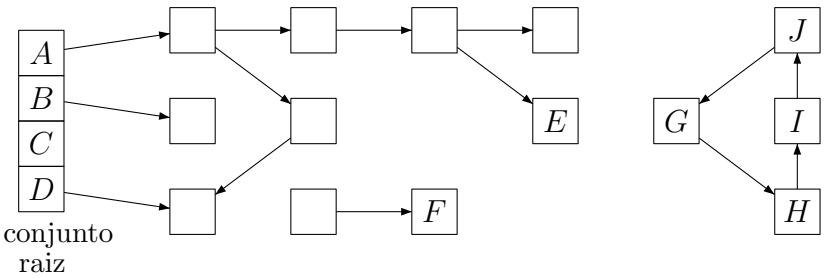

Figura 2.1: Grafo de conectividade.

Após determinar quais objetos são acessíveis, o coletor precisa determinar quais objetos são coletáveis, e efetivamente desalocar a memória correspondente. Note que encontrar a localização (o endereço) dos objetos coletáveis através do grafo de conectividade não é necessariamente uma tarefa trivial, e dependendo da implementação, pode demandar um esforço adicional de processamento não desprezível.

Coletores que empregam técnicas de rastreamento funcionam em *ciclos*. Enquanto o coletor constrói o grafo de conectividade e efetua a desalocação de obje-

tos, a execução do programa cliente é interrompida[1]. Diferentemente de coletores que utilizam contagem de referências, coletores baseados em rastreamento só são capazes de determinar se um objeto é efetivamente coletável após o término da fase de rastreamento de cada ciclo de coleta. Transcorre portanto um intervalo de tempo indeterminado entre o momento em que a última referência para um objeto qualquer é destruída, e o coletor efetivamente identifica este objeto como coletável.

2.1 Contagem de Referências

Contagem de referências [25] é provavelmente a técnica mais simples para efetuar a reciclagem automática de memória. Esta solução consiste em associar um contador de referências a cada objeto alocado na memória heap[2]. Sempre que uma nova referência para um objeto é criada, o contador deste objeto é incrementado. Analogamente, sempre que uma referência para um objeto é destruída, o seu contador é decrementado. Quando a contagem de referências atinge zero, os campos do objeto são examinados em busca de ponteiros. A contagem de referências de cada objeto referenciado por estes ponteiros é decrementada de forma recursiva, e a memória ocupada pelo objeto original, e por qualquer objeto cuja contagem de referências também atingiu zero, é desalocada.

Além da simplicidade, outra propriedade atraente do mecanismo de contagem de referências é a sua natureza incremental. A carga de processamento associada à coleta de lixo é distribuída de forma relativamente homogênea, e é intercalada com a execução do programa cliente. Em sistemas que demandam processamento em tempo real, essa característica é bastante importante.

A despeito dessas vantagens, dois problemas impedem a adoção ampla de contadores de referências como o mecanismo básico de gerenciamento de memória: tanto a eficácia quanto a eficiência de coletores que empregam esta solução são relativamente limitadas.

Não é possível, exclusivamente através de contagem de referências, coletar objetos que formam estruturas contendo referências cíclicas. Considere por exemplo o ciclo de referências formado pelos objetos G, H, I, J na Figura 2.1. Apesar

[1] Em coletores incrementais a construção do grafo de conectividade pode ser feita de forma intercalada com a execução do programa cliente. Mas durante a fase de coleta a execução deste último precisa ser suspensa.

[2] O contador geralmente é representado por um byte extra inserido no header de objetos alocados na memória heap.

destes objetos não poderem ser acessados pelo programa cliente (não podem ser alcançados a partir do conjunto-raiz), as referências existentes entre os mesmos impede que a contagem de referências chegue a zero. Esse tipo de situação representa um vazamento de memória, e só pode ser evitado com o auxílio de técnicas de rastreamento.

Para controlar a contagem de referências de objetos é necessário que o compilador introduza um bloco de código adicional para quase todas as operações de atribuição, chamadas de funções e saídas de blocos existentes no código original do programa. Uma operação simples de atribuição entre duas variáveis, por exemplo `a = b`, pode dar origem a um código como

```
if (a!=b){
  synchronize(b){
    b.refCount++;
  }
  a.decrementCount();
  a = b;
}
```

onde `decrementCount` é definido como

```
void decrementCount(){
  // verifica se o objeto deve ser desalocado
  if (refCount == 1){
    // obtém todos os objetos diretamente referenciados
    // por este objeto
    Object child[] = getChilds();
    // decrementa a contagem de referências dos filhos
    // de forma recursiva
    for (int i = 0; i < child.length; i++)
      child[i].decrementCount();
    // desaloca a memória associada a este objeto
    free(this);
  }  else
    synchronize(this){
      --refCount;
    }
}
```

O número de instruções inseridas para atualizar a contagem de referências é diretamente proporcional ao tamanho do programa, com uma constante de proporcionalidade significativa. Como consequência, quando comparada a outras técnicas de coleta de lixo, contagem de referências apresenta um desempenho geral menos favorável.

2.2 Mark and Sweep

Um ciclo de coleta de lixo em um sistema que emprega a técnica *mark-and-sweep* [47] pode ser dividido em duas fases distintas:

- A primeira fase (marcar) consiste em percorrer o grafo de conectividade, marcando todos os objetos encontrados na memória heap que são acessíveis a partir do conjunto-raiz. A marca pode fazer parte da estrutura do objeto (geralmente um bit no header de qualquer objeto alocado na memória heap), ou ser representada através de uma estrutura de dados externa (um bitmap), mantida pelo coletor ou pelo runtime.

- A segunda fase (varrer) consiste em percorrer toda a memória heap disponibilizada para a aplicação. A memória associada aos objetos que não foram marcados é simplesmente desalocada. Visando o próximo ciclo de coleta, os objetos marcados tem a marca apagada (novos objetos são sempre alocados com a marca apagada).

Essa dinâmica apresenta dois problemas que podem afetar negativamente o desempenho de aplicações. O primeiro problema está associado à fragmentação da memória. Como a memória livre (que foi reciclada) encontra-se fisicamente intercalada com blocos que representam objetos ativos, depois de um período de tempo torna-se difícil ou até impossível alocar objetos maiores, mesmo havendo uma quantidade total de memória livre suficiente. O segundo problema diz respeito ao custo de processamento associado à coleta de lixo. Em cada ciclo de coleta toda a memória heap precisa ser percorrida, independentemente do número de objetos acessíveis (o custo de coleta é linear em relação ao tamanho da memória disponibilizada para o programa cliente).

2.3 Mark Compact

Coletores *mark-compact* também efetuam a coleta de lixo em duas fases:

- Na primeira fase é feito o rastreamento dos objetos de maneira idêntica a coletores mark-and-sweep: a partir do conjunto-raiz é percorrido todo o grafo de conectividade, marcando-se os objetos encontrados.

- Na segunda fase o coletor percorre a memória heap e mapeia todos os objetos marcados (basicamente o coletor registra o endereço e o tamanho de todos os objetos marcados, e a partir destas informações calcula o novo endereço que cada objeto irá ocupar após a compactação da memória). Em seguida os objetos ativos são movidos (compactados) para o início da memória heap, geralmente seguindo a ordem de endereçamento lógico. Ao mesmo tempo, as referências para os objetos, tanto aquelas contidas no conjunto-raiz quanto as encontradas nos objetos movidos, são atualizadas com os novos endereços.

Esse esquema garante a manutenção de um bloco contíguo de memória livre, e evita portanto o problema de fragmentação. Em compensação são necessárias múltiplas varreduras de toda a memória heap em cada ciclo de coleta, o que representa um custo de processamento relativamente alto.

2.4 Stop and Copy

Coletores *stop-and-copy* [29, 22] conseguem reduzir o tempo total de um ciclo de coleta ao custo de uma utilização maior da memória. Neste tipo de coletor a memória heap é dividida em dois blocos distintos (denominados *semi-espaços*) de tamanho igual: o *fromspace* e o *tospace* (ver a Figura 2.2).

Objetos novos são alocados no fromspace enquanto houver espaço disponível no mesmo. Quando não for possível efetuar novas alocações, um ciclo de coleta é disparado.

Diferentemente de coletores mark-and-sweep e mark-compact, a coleta é efetuada em uma única fase. O rastreamento ocorre através da varredura do grafo de conectividade, mas quando um objeto é encontrado, ele é imediatamente copiado para o tospace (Figura 2.3), e em seu lugar é instalado um ponteiro contendo o novo endereço do objeto recém-copiado (o *forwarding address*). A atualização dos endereços é feita de forma simples e acontece na medida em que os objetos são encontrados e copiados. Após o grafo de conectividade ter sido totalmente varrido e todos os objetos acessíveis terem sido copiados, o tospace torna-se o fromspace, e vice-versa.

É fácil perceber que neste esquema a coleta dos objetos inacessíveis acontece

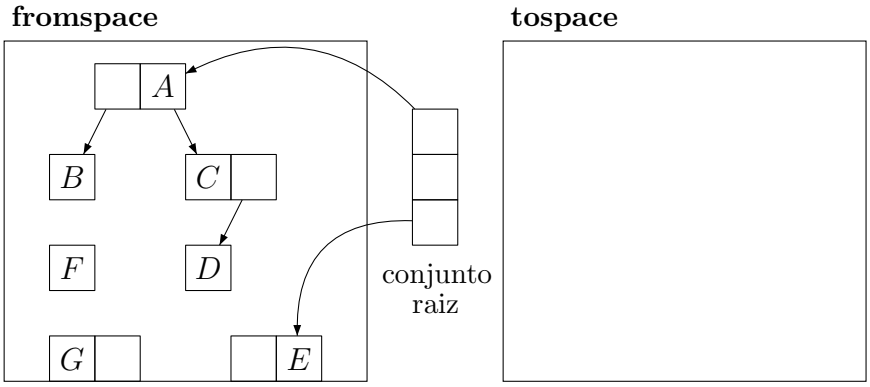

Figura 2.2: Coletor stop-and-copy antes da coleta.

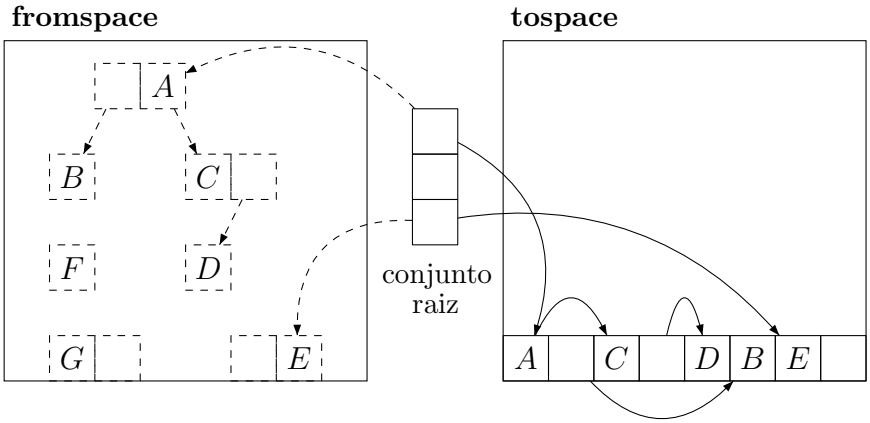

Figura 2.3: Coletor stop-and-copy depois da coleta.

de forma implícita, quando os semi-espaços são invertidos. Logo não é necessário determinar o endereço dos objetos coletáveis.

O custo de processamento de coletores stop-and-copy é proporcional ao número de objetos ativos na memória, o que no caso geral é bem menor do que o custo associado às outras técnicas apresentadas até aqui. Além disso a forma como os objetos são movidos (seguindo a cadeia de referências entre os mesmos), gera uma melhor localidade, proporcionando ganhos adicionais de desempenho.

2.5 Coletores Generacionais

Coletores generacionais [46] utilizam uma técnica de coleta semelhante à stop-and-copy: os objetos encontrados durante a varredura do grafo de conectividade são movidos entre semi-espaços distintos, e a coleta de objetos inacessíveis acontece de maneira implícita. Porém, esses coletores conseguem obter um desem-

penho ainda melhor ao segregar objetos de acordo com a sua idade relativa, e efetuar coletas seletivas (alguns ciclos de coleta restringem-se a partes reduzidas da memória heap).

Estudos empíricos indicam que a maioria dos objetos usados por uma aplicação qualquer, independentemente da linguagem de programação considerada, pode ser desalocada rapidamente [64]. Apenas um número pequeno de objetos é usado durante todo o ciclo de vida da aplicação. Como coletores stop-and-copy copiam todos os objetos acessíveis em cada ciclo de coleta, objetos perenes acabam sendo copiados inúmeras vezes, incorrendo-se assim em um custo de processamento relativamente alto e desnecessário.

Coletores generacionais evitam grande parte desse custo ao usar uma política de coleta diferenciada. Neste tipo de coletor cada semi-espaço corresponde a uma *geração*, e as gerações são ordenadas de acordo com a idade relativa dos objetos que contém. O número de gerações (n, onde $n \geq 2$), e a política de cópia de objetos entre gerações (*promoção*), depende da implementação específica, mas a alocação e a coleta de objetos seguem basicamente as seguintes regras:

- Novos objetos são sempre alocados na geração mais jovem (geração zero ou *berçário*).
- Gerações mais jovens são coletadas com maior frequência.
- Quando uma geração qualquer é coletada, todas as gerações mais jovens também são coletadas.
- Objetos que sobrevivem a um número pré-determinado de coletas de uma geração g, onde g é menor do que n, são promovidos (copiados) para a geração $g + 1$.

Para evitar que objetos alocados imediatamente antes do início de um ciclo de coleta sejam sempre promovidos, geralmente a promoção de objetos no berçário é feita somente se o objeto sobreviver a pelo menos dois ciclos de coleta. Para as demais gerações, o critério mais comum de promoção é sobreviver a um único ciclo de coleta.

A coleta de objetos na geração n pode ser feita através de qualquer uma das técnicas baseadas em rastreamento que discutimos. O uso de stop-and-copy, onde a geração n é dividida em dois semi-espaços distintos, é interessante na medida em que possibilita a reutilização das rotinas do coletor de lixo.

Não é difícil perceber que para coletar um subconjunto de gerações de forma independente é preciso conhecer todas as referências acessíveis que apontam para os objetos contidos nestes semi-espaços. Assim, mesmo restringindo a área de coleta, a princípio seria necessário efetuar um rastreamento completo da memória para poder determinar quais objetos são acessíveis.

Algumas técnicas permitem efetuar a coleta de um subconjunto de gerações sem efetivamente rastrear todas as gerações mais antigas. Uma opção é manter listas externas contendo referências para todos os objetos que apontam para objetos localizados em gerações mais novas. Se uma atribuição fizer com que um objeto de uma geração mais antiga passe a apontar para um objeto localizado em uma geração mais nova, uma nova referência é criada na lista associada a geração mais nova (estas listas são conhecidas como *remembered sets* [64])[3]. Quando um subconjunto de gerações é coletado, o grafo de conectividade é construído a partir das referências do conjunto-raiz que apontam para estas gerações, e das referências encontradas em seus respectivos remembered sets. A atualização dos remembered sets pode ser feita através de barreiras de escrita, o que limita o custo de processamento adicional associado a essa tarefa.

Por fim, vale observar que as implementações padrão tanto de Java quanto C# empregam coletores generacionais, mas utilizam técnicas um pouco mais sofisticadas do que aquelas descritas aqui para a promoção de objetos [3, 56].

[3]Não são permitidas referências a partir de um objeto de uma geração mais nova para um objeto de uma geração mais antiga. Para garantir esta invariante, sempre que for atribuído um objeto de uma geração mais antiga para um objeto de uma geração mais nova, este objeto é imediatamente transferido para a geração mais antiga.